

# MIMOSA

## Microsystems platform for Mobile Services and Applications

**FP6 Contract: IST-2002-507045**

---



### **WP2 – System architecture**

#### **Deliverable report**

Deliverable ID:	<b>D2.1[2]</b>
Deliverable Title:	<b>Overall MIMOSA architecture specification (OMAS)</b>
Responsible partner:	Nokia
Contributors:	Antti Lappeteläinen, Heikki Nieminen Mikko Vääräkangas, Hannu Laine, Dirk Trossen, Dana Pavel

#### PROPRIETARY RIGHTS STATEMENT

This document contains information, which is proprietary to the MIMOSA Consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with prior written consent of the MIMOSA consortium.

## Document Information

**Document Name:** Overall MIMOSA architecture specification (OMAS)  
**Document ID:** MIMOSA-WP2-D2.1[2]  
**Revision:** 0.99  
**Revision Date:** 15/06/05  
**Author:** Dirk Trossen  
**Security:** Public (PU)

## Approvals

	Name	Company	Date	Visa
<i>Technical Coordinator</i>	Pascal ANCEY	ST Fr	24/11/05	OK
<i>WP leader</i>	Vladimir ERMOLOV	Nokia	15/06/05	OK
<i>Quality Manager</i>	Cédric ROBET	ALMA	18/06/05	OK

## Documents history

Revision	Date	Modification	Author
0.1	16-March-03	MIMOSA template taken into use	Antti Lappeteläinen
0.2	12-May-04	4. Terminal and 5. radio sensor node architectures generalized	Mikko Vääräkangas
0.21	14-May-04	3. Mimosa architecture target added and few corrections made	Mikko Vääräkangas
0.9	June 15 <sup>th</sup> -04	Proposal for MIMOSA project partners	Antti Lappeteläinen
0.91	June 23 <sup>rd</sup> -04	Chapter 6 update	Mikko Vääräkangas
0.xx	December 15 <sup>th</sup> -04	First integration of the remote connectivity	Dirk Trossen
0.95	June 8 <sup>th</sup> 05	Added appendix for remote connectivity middleware spec	Dirk Trossen
0.99	June 9 <sup>th</sup> 05	Editorial updates throughout the document	Dirk Trossen

- 1. INTRODUCTION..... 5**
- 2. MIMOSA ARCHITECTURE IN RELATION TO THE MIMOSA VISION..... 5**
- 3. OVERALL MIMOSA ARCHITECTURE ..... 6**
  - 3.1. OVERALL MIMOSA DEVICE ARCHITECTURE ..... 6
  - 3.2. LOCAL MIMOSA DEVICE ARCHITECTURE ..... 7
- 4. TERMINAL ARCHITECTURE..... 9**
  - 4.1. HOST ARCHITECTURE ..... 9
    - 4.1.1. Hardware Functionality ..... 10
    - 4.1.2. Software functionality..... 11
    - Middleware..... 11
  - 4.2. INTERFACE BETWEEN HOST AND MIMOSA LOCAL CONNECTIVITY MODULE ..... 11
  - 4.3. HOST SENSOR INTERFACE..... 12
    - 4.3.1. Physical interface..... 12
    - 4.3.2. Drivers..... 13
    - 4.3.3. Sensor interface drivers..... 13
    - 4.3.4. Simple Sensor Interface (SSI) protocol ..... 15
  - 4.4. INTERFACE HOST – UI HW ..... 15
  - 4.5. APPLICATION INTERFACE ..... 15
  - 4.6. POWER INTERFACE ..... 16
  - 4.7. REMOTE CONNECTIVITY INTERFACE ..... 16
  - 4.8. MIDDLEWARE FUNCTIONALITY ..... 17
- 5. SENSOR RADIO NODE ARCHITECTURE ..... 17**
  - 5.1. HOST ARCHITECTURE ..... 17
    - 5.1.1. Hardware functionality ..... 18
    - 5.1.2. Software functionality..... 18
  - 5.2. INTERFACE HOST – SENSOR ..... 19
    - 5.2.1. UART ..... 20
    - 5.2.2. I2C ..... 20
    - 5.2.3. SPI ..... 20
    - 5.2.4. Analog interface ..... 21
  - 5.3. INTERFACE HOST –CONNECTIVITY..... 22
  - 5.4. INTERFACE HOST – ENERGY SCAVENGING ..... 23
  - 5.5. INTERFACE BATTERY - ENERGY SCAVENGING ..... 23
  - 5.6. POWER INTERFACE ..... 23
- 6. TAG ARCHITECTURE..... 25**
  - 6.1. RFID SENSOR ARCHITECTURE ..... 25
  - 6.2. SENSOR INTERFACE ..... 26
  - 6.3. FUNCTIONAL DESCRIPTION ..... 26
  - 6.4. SENSOR BASIC FEATURES ..... 27
- 7. REFERENCES ..... 27**
- 8. APPENDIX: REMOTE CONNECTIVITY MIDDLEWARE SPECIFICATION..... 28**
  - 8.1. FUNCTIONALITY OF EACH COMPONENT..... 28
    - 8.1.1. Event Communication Paradigm in REMONT..... 29
    - 8.1.2. Event Delivery Component ..... 30
    - 8.1.3. Acquisition Component..... 30
    - 8.1.4. Query Resolver Component ..... 31
    - 8.1.5. Aggregation Component..... 32
    - 8.1.6. Access Control Component ..... 32
    - 8.1.7. Storage Component..... 32

- 8.1.8. Registration & Availability Component ..... 33
- 8.2. INTERFACES BETWEEN THE COMPONENTS..... 33
  - 8.2.1. The RAPI Interface (Remote Connectivity API)..... 33
  - 8.2.2. The RAPI<sub>ARA</sub> interface..... 34
  - 8.2.3. The RAPI<sub>AA</sub> interface ..... 34
  - 8.2.4. The E<sub>D</sub> Interface..... 35
  - 8.2.5. The R<sub>A</sub> Interface..... 35
  - 8.2.6. The A<sub>C</sub> Interface..... 36
  - 8.2.7. The SSI<sub>A</sub> Interface ..... 36
  - 8.2.8. The SSI<sub>R</sub> Interface ..... 37
  - 8.2.9. The E<sub>D(local)</sub> Interface ..... 37
  - 8.2.10. The A<sub>RA(local)</sub> Interface ..... 37
  - 8.2.11. The A<sub>ctrl(local)</sub> Interface ..... 38
  - 8.2.12. The A<sub>A(local)</sub> Interface ..... 38
  - 8.2.13. The A<sub>Q(local)</sub> Interface..... 38
  - 8.2.14. The A<sub>S(local)</sub> Interface ..... 39

## 1. INTRODUCTION

The deliverable is part I of Overall MIMOSA Architecture Specification, OMAS. The OMAS defines an open interface architecture that enables trials and demonstrations of innovative hardware and software modules related to ambient intelligence. The focus areas of OMAS are to provide platform to which new local connectivity, sensors and UI hardware modules can be integrated and tested in real application environment. For application development OMAS offers scalable computing platform and an interconnection to alternative UI devices. OMAS is not targeted as a product platform.

Chapter 2 introduces MIMOSA vision and concludes the needed device types. Chapter 3 presents the overall MIMOSA architecture and the key interfaces. Chapter 4 describes the architecture of MIMOSA user terminal, whereas chapters 5 and 6 outline the same for sensor radio nodes and RFID tags, respectively.

## 2. MIMOSA ARCHITECTURE IN RELATION TO THE MIMOSA VISION

In the MIMOSA vision, the personal mobile phone is chosen as the trusted intelligent user interface to Ambient Intelligence and a gateway between the sensors, the network of sensors, the public network and the Internet. MIMOSA develops and implements an open **technology platform architecture** for such vision. Further, MIMOSA aims at developing a remote connectivity platform that allows for application servers in the Internet to apply locally obtained sensor data (using the MIMOSA host architecture) within the context of specific services and applications. However, the services themselves are not within the scope of MIMOSA.

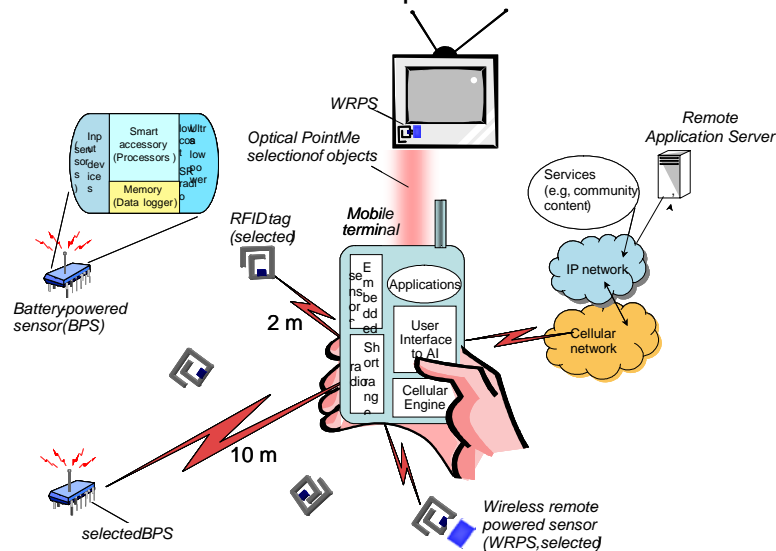


Figure 1: Architecture basis for local connectivity between the handset and the environment

The key target of the MIMOSA architecture is to provide an architecture framework for an open demonstration platform that can be used to demonstrate different MIMOSA made and commercial hardware within the contexts defined by the MIMOSA applications. The architecture of the MIMOSA platform can best be described by using top down approach. At the top level MIMOSA platform can be

divided into four physical entities and wireless interfaces between the entities. The physical entities are illustrated in Figure 1.

The five entities are Mobile Terminal, Remote Application Server, Sensor Radio Node, Active RFID node and Passive RFID TAG. Active and passive TAGs will have common OMAS architecture. The usage of optional external power supply will determine whether a TAG is active or passive.

The following chapters present a high-level description of the five entities and their external and internal interfaces.

### 3. OVERALL MIMOSA ARCHITECTURE

The target for the design of the overall MIMOSA architecture is to provide fast, flexible and comprehensive platform for combine new connectivity, sensor and UI hardware and novel applications to new ensemble. Architecture is not targeted to products as it is, but rather to give fast test and development possibilities when architecture and platform is ready while applications and hardware are under development. Architectural platform is also intended to be as reusable as possible. Architecture will give normative specifications for interfaces and directive requirements for different predefined functional blocks in devices.

#### 3.1. Overall MIMOSA Device Architecture

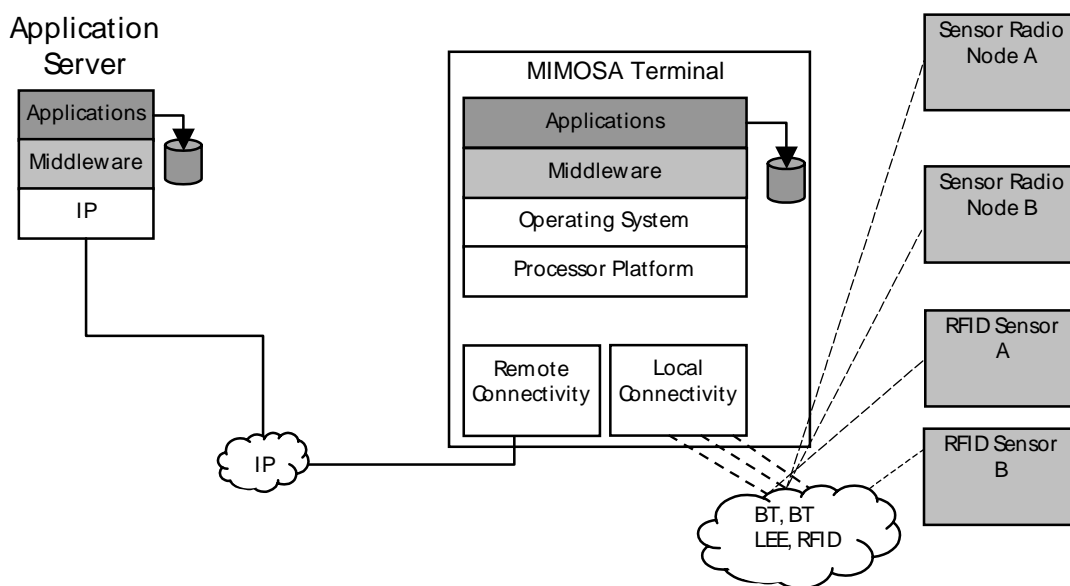


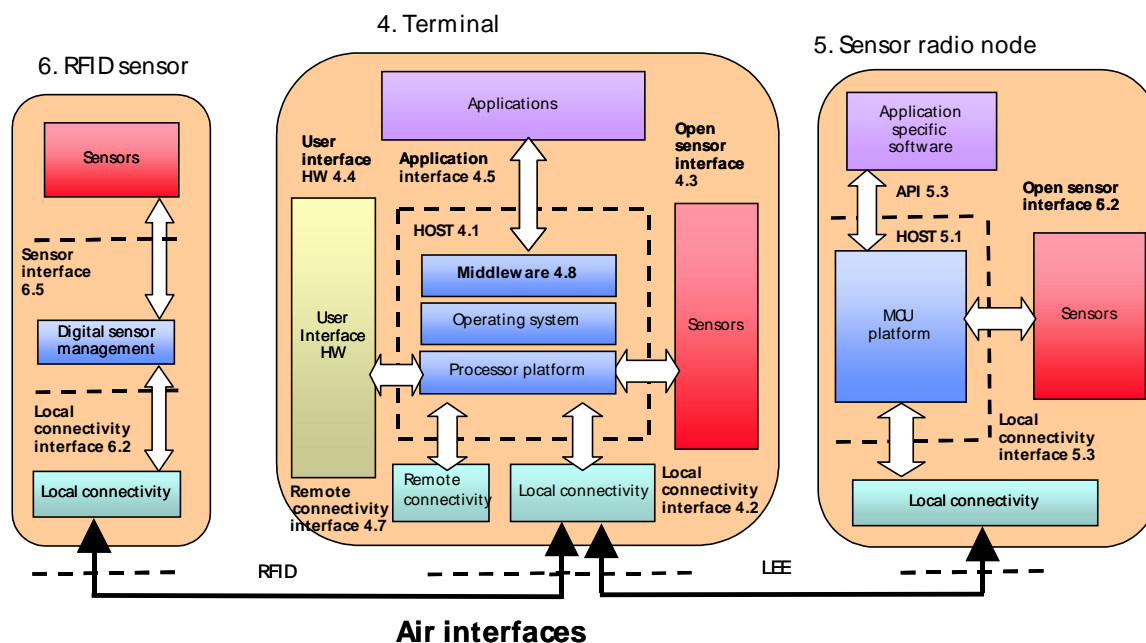
Figure 2: Overall MIMOSA Device Architecture

The overall MIMOSA device architecture is shown in Figure 2. It resembles four major devices, namely the remote application server, the MIMOSA terminal, and RFID node as well as sensor radio node as the local sensing devices. The remote application server does not need to exist in all application scenarios, however needs integration in the overall architecture due to the required remote connectivity in the MIMOSA terminal. The remote application server as well as the MIMOSA terminal

will run the middleware component as a common software platform to be accessed by terminal-local applications or remote application (executed within the remote application server).

### 3.2. Local MIMOSA Device Architecture

The Local MIMOSA device architecture has three different entities as building blocks of ambient intelligence: Terminal, sensor radio node and RFID sensor, which can be either battery powered or remote powered. All of these entities have local connectivity capability used to combine all of the different systems to ambient intelligence around user. If required by the use case, remote connectivity is provided by the terminal to connect to remote application servers for further functionality. Intelligence is composed of applications, the middleware, and sensors. Collected information will be processed to usable format and the terminal translates user



behavior so that environment can reply as expected.

**Figure 3: Local MIMOSA device architecture**

Each of the MIMOSA entities has several functional blocks connected to each other via multiple interfaces. All of the entities have own chapter in this document. Chapter numbers are shown in Figure 3. Specified interfaces are also specified in this figure.

Terminal device is the trusted device of the user. Most important blocks of the terminal are: processor platform, local connectivity module, user interface HW and embedded sensors. Processor platform is running an operating system. A middleware component is provided that provides to the application commonly used functionality, such as provisioning of sensor data, abstracting sensor-specific formats, providing local aggregation functionality and support discovery of surrounding sensors. The application uses the middleware functionality to implement ambient intelligence applications. Remote applications, using the terminal as a sensing gateway, can be implemented in a remote application server (see Figure 2). User interfaces are for interaction between user and the ambient intelligent. Applications

will handle all information collection, processing and displaying of the results through the middleware functionality. Local connectivity module offers connection to all surrounding MIMOSA sensors devices from terminal. There can also be some embedded sensors on the terminal platform to enable sensing ability for user interfaces or some application. An example implementation for communication protocol between applications and different sensors is SSI, simple sensor interface, which is described briefly in 4.3.4 and more extensively in MIMOSA sensor architecture specification [3]. Remote connectivity over cellular or other wireless access technology is used to connect to optional remote application servers.

The OMAS terminal architecture should enable trails of RFID readers with different specifications and to benchmark them by using real application environment. In addition the terminal architecture should enable performance and usability comparison of RFID and low-power radio solutions. On the ambient user interface side the architecture should enable possibility to easily add various UI and sensor HW in order to test and demonstrate how terminal device can be used as interface to ambient surroundings.

Radio sensor node consists of micro-controller based host, local connectivity module and sensors. Micro-controller has processor core that will enable application specific software running on the sensor node this includes communication protocol stack and sensor information processing. Pre-processing capability in sensor radio node, can give high-level sensor information for user applications running on the terminal. Radio sensor node has own radio, so it can advertise services that are available independently and communication range is longer than RFID based systems.

The OMAS radio sensor node architecture should provide platform for testing various low power radio solutions and distributed computing architectures for processing of sensor information. The second target of the architecture is to provide platform for testing of different energy scavenging technologies as a technology alternative to remote powered.

RFID sensor is very integrated sensor system. Local connectivity is most important part of this system. The communication protocol stack and sensor management capability are only criteria for RFID sensor. There is two different lower level interfaces on the RFID sensor: air interface and sensor – local connectivity interface. Both of these are specified in chapter 6 as shown in Figure 3.

The OMAS RFID sensor architecture should create together with the terminal architecture an ultimate platform for demonstrating the capabilities of various RFID technologies and low-power sensors create. The local connectivity architecture specific issues are addressed in

A reference MIMOSA architecture implementation will be presented later in MIMOSA system integration specification. This deliverable will be finalized in June 2005. It consists of specification of the used hardware and software that are needed, reference implementation, and interface description for new hardware and applications that are used on presented platform.

## 4. TERMINAL ARCHITECTURE

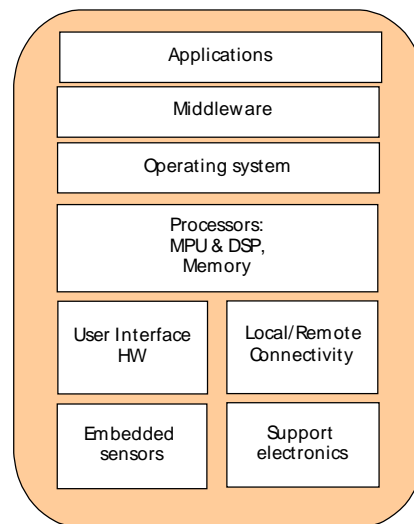
Terminal device is user's trusted device that is designed to be interface between environment and user. Terminal device has data processing power to run applications, which will deliver new information to user. That information is collected from environment, including the user, using embedded sensors in terminal device or distributed sensors around. Connection to external distributed sensors is done using MIMOSA local connectivity module.

The high level terminal architecture is divided in to host architecture and interface architectures. The following interfaces are defined to be important for MIMOSA architecture: host - local connectivity, host – sensor, host –UI and power interface.

### 4.1. HOST ARCHITECTURE

The host architecture is not in the focus of the MIMOSA project. The host should provide processing, memory and interface capabilities similar to the existing and near future smart phone platforms. The hardware of the host platform should be capable to support several operating systems, including Linux and Symbian. Host will serve as processor platform that is used to demonstrate MIMOSA environment and applications.

Target is a flexible host that delivers easy development platform to both hardware and software development. This claim expects that platform has multiple interfaces to different kind of peripheral devices, both wireless and ordinary connections. There should be enough processing power to allow complicated data processing and rather large memory for data storage purposes. This all has to be usable packet with enough expandability and flexibility for future use also.



**Figure 4: Overall architecture of MIMOSA terminal device**

The host architecture can be divided to two parts: hardware and software. Figure 4 shows overall architecture of the terminal used in MIMOSA project. The host architecture includes all blocks excluding local connectivity, embedded sensors and specific user interface devices as display and keyboard.

The MIMOSA host consists of a processor system that includes processors, memories and necessary peripheral devices. The target is to provide a platform for application and external hardware development. The host has an operating system running and it will include necessary drivers so that basic hardware interfaces is ready for use as specified. It will also implement a middleware the implements commonly used functionality, such as provisioning of sensor data, abstracting sensor-specific formats, providing local aggregation functionality and support discovery of surrounding sensors. Applications will be build on top of this middleware to re-use this commonly used functionality. All external devices can be used from applications if used interface is supported. An optional remote connectivity capability within the MIMOSA terminal device, together with the middleware component, can be used for applications running in a remote application server, which has more computing capability.

The interface between the host and UI hardware can in some cases be distributed between several devices. This could be needed, for instance, when a laptop with a bigger CPU is needed to run a heavier application, or if a mobile phone is used as the application platform. One connection option is Bluetooth, depending on the UI hardware. The interface 4.5 from the host to the applications can also be distributed. In such a case the protocols over this API are simply ported on top of the UI hardware interface 5.4. The protocol towards the applications must be specified in further work.

#### 4.1.1. Hardware Functionality

Host hardware platform consists of processing unit that has a microprocessor unit (MPU) with memory management unit (MMU) and a digital signal processor (DSP). Desired processor is ARM9, but architecture can be implemented also on other platform. Processors will need a common volatile memory block and a nonvolatile memory which is used to store binary codes of the operating system and applications with user information. Good amount of memory can be 64 MB volatile RAM memory and 16 MB nonvolatile Flash memory. A comprehensive set of peripheral interfaces is needed, at least JTAG, UART, SPI, I<sup>2</sup>C and  $\mu$ Wire. User interface (UI) devices should also have their own interfaces and peripheral controllers if needed. Most important UI devices are keyboard, LCD display, touch panel and microphone. All previously mentioned features are obligatory and processor platform has to have them all.

There should also be a lot of general-purpose i/o-pins available for MIMOSA interfaces. Specific interfaces are host - local connectivity, host – UI and host – battery management. A commercially available host platform is listed in [1].

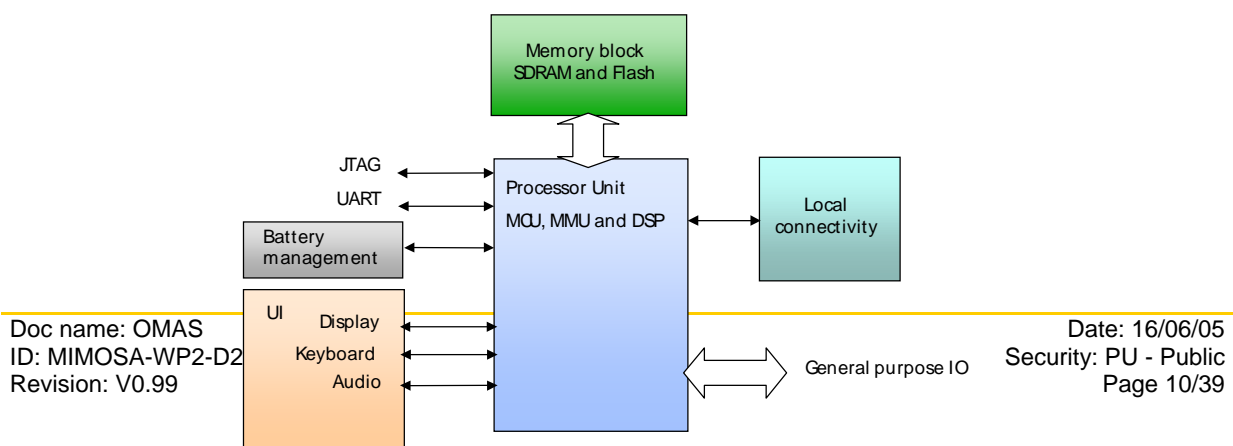


Figure 5: Block diagram of host

#### 4.1.2. Software functionality

##### Operating systems

The host platform has two processor cores. ARM9 is preferred as MPU but DSP is optional. The onboard memory of 64 MB is sufficient for hosting large operating systems for the MCU. The best candidates for the operating system are Symbian and Linux. Since Linux is better suited for multipart research, it will be used, at least at the beginning. However, the MIMOSA platform should not be restricted to a single operation system. The OMAS architecture does not specify the DSP operating system.

##### Software development environment

For Linux version of OMAS will enable GNU tools based application development. If application utilizes DSP, a commercial development environment is required.

Symbian OS related requirements would be fixed later.

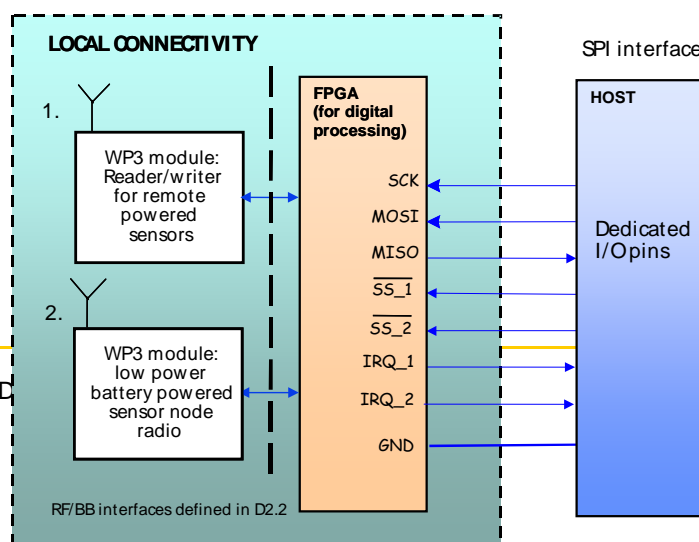
##### Middleware

The middleware abstracts commonly used functions from the applications, enabling the re-use of these commonly used functions, e.g., for discovery and acquisition of sensor values. The middleware implements the different APIs for remote and local connectivity, context awareness and UI, as outlined in the particular work packages (WP2.2, 2.3, 2.4 and 2.5) and as specified in work package 2.7.

#### 4.2. Interface between Host and MIMOSA Local Connectivity Module

##### Physical interface

Figure 6 presents the high-level interface between MIMOSA local connectivity and terminal host. More precisely, the MIMOSA local connectivity module features FPGA (details will be fixed later), which will be connected to available the I/O connector of the host using enhanced SPI “Serial Peripheral Interface” protocol. SPI is developed for on board inter peripheral communications. The SPI enhancements relate to power



consumption optimization. Update Host acts as the master of SPI protocol. Hence the host is responsible of generated the clock signal that is used to synchronize each bit in the both transmission directions, MISO and MOSI. The clock frequency will be fixed later.

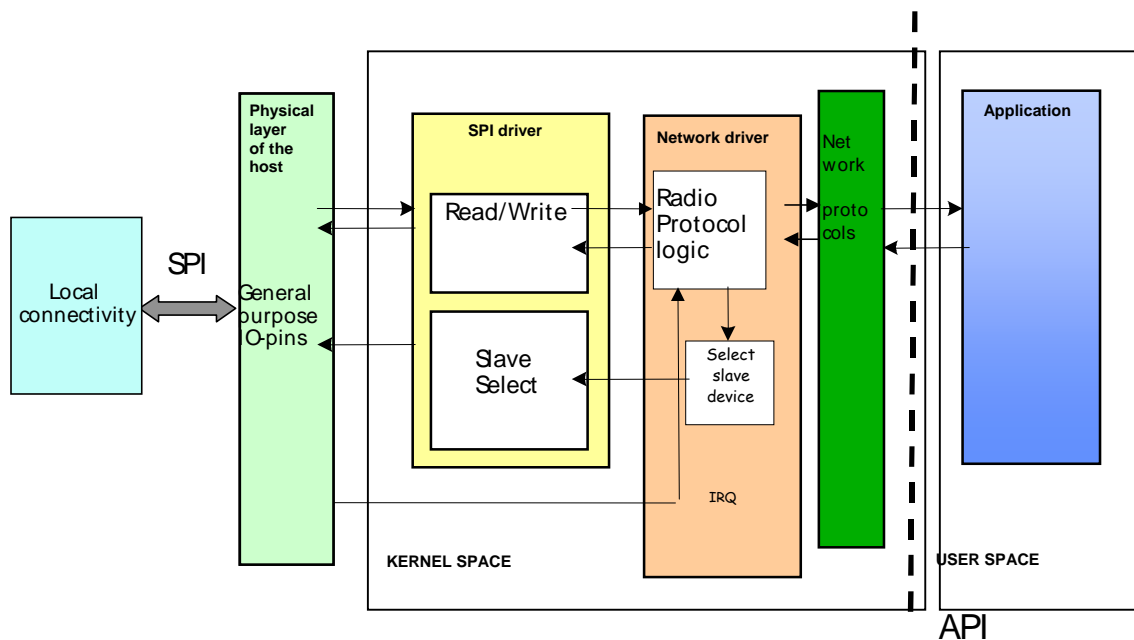
**Figure 6: High-level MIMOSA local connectivity architecture for terminals**

### Driver

Figure 7 presents the driver architecture. MIMOSA local connectivity drivers are divided in two parts. Lower layer SPI driver is hides most hardware dependent issues from the network driver. SPI driver provides read and write services, 1 byte at a time, along with slave select possibility to upper layers.

Network driver will have the ability to drive the MIMOSA local connectivity modules. This includes data exchange and control. For transmitting data the driver provides MIMOSA\_TX function for the kernel, see API section. For receiving data from the MIMOSA local connectivity module the driver provides interrupt handler function for the kernel.

User or application can control the behavior of the driver via *ioctl* function calls. The function calls are fixed later based on demonstration needs.



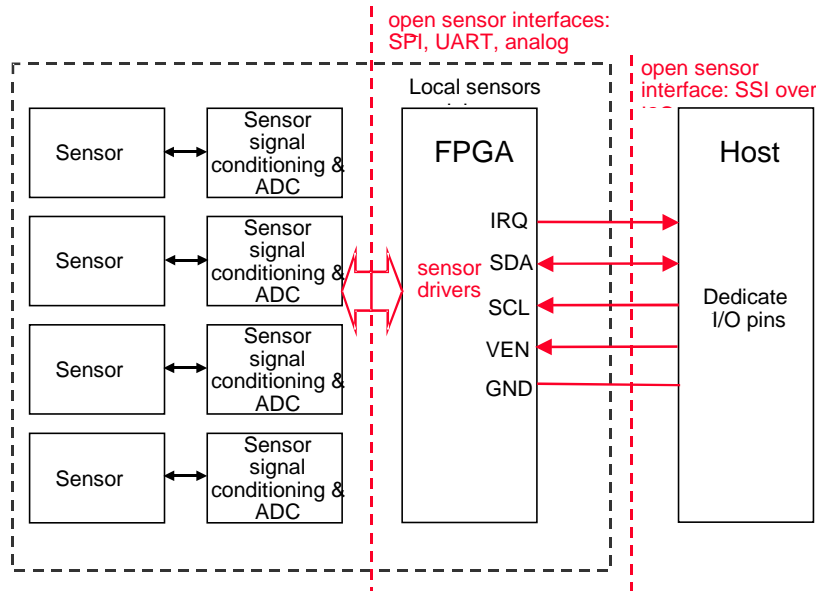
**Figure 7: MIMOSA Connectivity drivers and SW interfaces for terminal**

## 4.3. Host sensor interface

### 4.3.1. Physical interface

Host typically has very limited amount of I/O pins. Therefore the sensors will not be connected directly to the host, but to the FPGA featuring sensor drivers. Figure 8 presents the high-level interface between MIMOSA local sensors module and MIMOSA terminal host. More precisely, the MIMOSA local sensors module features FPGA (details will be fixed later), sensors that have digital output, and AD converter for analog sensors. FPGA will be connected to available I/O connector of the host

using enhanced I2C protocol. I2C modifications are related to power consumption optimization. They are; dedicated interrupt (IRQ) possibility from MIMOSA local sensors module to the host and power on/off (VEN) switch by the host. The host acts as the master of I2C protocol. Hence the host is responsible of generated the clock signal that is used to synchronize each bit in the both transmission directions. The clock frequency will be fixed later.



**Figure 8: High level interface between MIMOSA local sensor unit and MIMOSA terminal host**

#### 4.3.2. Drivers

The drivers are divided into two layers. First there are the interface drivers that handle the hardware dependent issues in communication between host and local sensors module and between local sensors module and sensors. Second layer is the device drivers. Device drivers for each sensor are in FPGA. The communication between local sensor module and host is handled by sensor module driver using simple sensor interface (SSI) protocol.

#### 4.3.3. Sensor interface drivers

**Table 1: I2C driver functions**

Data Interface	Definition
S_init_I2C(void)	Routines for initializing the HW I2C interface
S_clock(void)	Send I2C clock to the bus
S_delay(void)	Delay function
S_wait (void)	Wait function
S_write_I2C(location, data)	This routine writes data to location using SW I2C implementation in microcontroller.
S_read_I2C(addr)	Reads data from address using I2C bus.

**Table 2: UART interface functions**

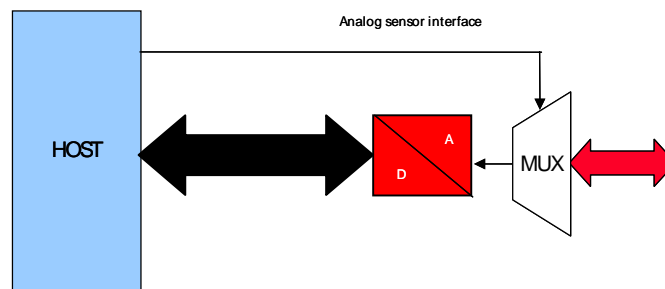
Data Interface	Definition
S_uart_init(mode, baud_rate)	Initializes UART (mode= UART, SPI, I2C)
S_uart_disable(void)	Disables UART
S_uart_putchar(c)	Puts a character to UART TX buffer and sends it out
S_uart_putstr(s)	Sends a string out through UART, returns 1 if success
S_uart_getchar(void)	Reads an incoming character from UART buffer
S_spi_getchar(void)	Reads an incoming character in spi mode
__interrupt void uart0_tx_isr(void)	UART TX data buffer empty interrupt
__interrupt void uart0_rx_isr(void)	UART receive interrupt routine, executed when incoming symbols are in RX buffer

**Table 3: SPI driver functions**

S_uart1_init(mode, baud_rate)	Initializes SPI (mode= UART, SPI, I2C)
S_uart1_disable(void)	Disables SPI
S_uart1_putchar(c)	Puts a character to SPI TX buffer and sends it out
S_uart1_putstr(s)	Sends a string out through SPI, returns 1 if success
S_uart1_getchar(void)	Reads an incoming character from SPI
S_spi1_putchar(c)	Sends a character out to SPI bus
S_spi1_getchar(void)	Reads a character in from SPI bus
S_spi1_getint(channel)	Reads an integer in from SPI bus
__interrupt void uart1_tx_isr(void)	SPI TX data buffer empty interrupt
__interrupt void uart1_rx_isr(void)	SPI receive interrupt routine, executed when incoming symbols are in rx buffer

### Physical interface for analog sensors

Several analog interfaces are available for sensors. Host has an AD conversion module and the analog inputs are multiplexed to AD converter. This block can be embedded straight to MCU chip or it can be external. Physical implementation differs, if this block is external, but driver functions have to be implemented as specified.



**Figure 9: Analog sensor interfaces**

**Table 4: AD converter block driver functions**

Data Interface	Definition
S_ad_init_sensor_n(void)	Configures AD channels for specified sensor_n

	measurement
S_ad_init_8(void)	Initialized AD converter for 8 channel measurement
S_adon (void)	Turn AD converter on
S_adoff (void)	Turn AD converter off
S_ad_start(void)	Start AD conversion
__interrupt void adc12(void)	AD converter interrupt (no interrupt/ADC overflow/ADC timing overflow)

#### 4.3.4. Simple Sensor Interface (SSI) protocol

SSI protocol is an open protocol. It is specifically designed for sensor systems. MIMOSA participant Suunto and sensor manufacturer Vaisala have participated into SSI definition work. It is the key tool for the plug and play operation that allows identification, control and data retrieval from sensor nodes regardless of their location. Table 5 summarizes the SSI protocol commands. Full details of the SSI protocol can be found in the appendix of [3].

**Table 5: Simple Sensor Interface protocol commands**

Command	Direction	Description
Q,q (0x51,0x71)	->	Query
A,a (0x41,0x61)	<-	Query reply
C,c (0x43,0x63)	->	Discover sensors
N,n (0x4E,0x6E)	<-	Discovery reply
Z,z (0x5A,0x7A)	->	Reset Wirsu device
G,g (0x47,0x67)	->	Get configuration data for a sensor
X,x (0x58,0x78)	<-	Configuration data response
S,s (0x53,0x73)	->	Set configuration data for a sensor
R,r (0x52,0x72)	->	Request sensor data
V,v (0x56,0x76)	<-	Sensor data response
D,d (0x44,0x64)	<-	Sensor response with one byte status
F,f (0x46, 0x66)	<->	Free data for custom purposes

#### 4.4. Interface host – UI HW

The host platform will use standard UI components, such as touch screen, speaker and microphone, which can be replaced with HW developed in Mimosasa. The Linux operating system contains the necessary software drivers. When new HW is added, the drivers must be updated or rewritten but at first there should be drivers for all used HW. In application programs, UI devices appear as typical Linux I/O devices. Further details are provided in the application programming environment documentation for selected host platform.

#### 4.5. Application Interface

The application interface defines the applications' access to host platform resources. This interface complements and builds upon the interface that the middleware of the host platform offers. The exact form of the interface is described on abstract level and will be specified more exactly in future versions.

The application interface offers access to local connectivity capabilities and sensors. Connectivity capabilities include discovering other devices (sensors and other

terminals), establishing connections, and closing down connections. Sensor access includes discovering and configuring sensors, obtaining data from sensors, and offering access to remote sensors, i.e., sensors connected to host terminal via wireless connection.

The application interface offers the required capabilities for forwarding sensor data to other terminals, for example using the SSI protocol over Bluetooth connection. Demonstration applications can therefore be developed using some other device for running the applications. For instance, the sensor data can be sent from the host terminal to a smart phone, which offers more capabilities for user interface and software development, and provides network access.

#### 4.6. Power interface

Terminal is targeted to be battery powered and it has few major parts that will need different voltage levels. Parts are processor platform, FPGA, sensors, communication devices and battery management and power electronics..

Figure 10 shows the architecture of the power block and interfaces to system and control logic. Amount of voltage levels is not fixed yet so we can add some if needed. Battery management interface has two digital signals that are used to indicate states of the power block: battery voltage dropping below preset battery low voltage level and charging of the battery.

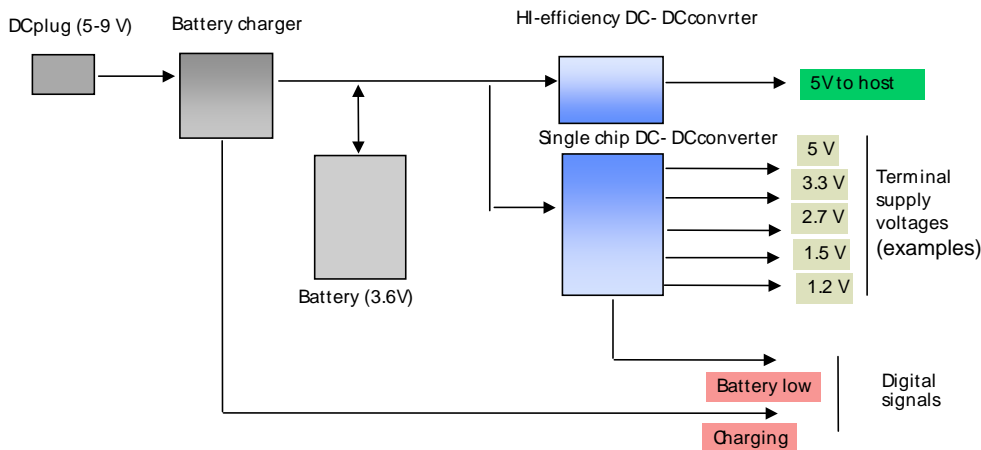


Figure 10: The power block of terminal device

#### 4.7. Remote Connectivity interface

The remote connectivity interface allows access to remote application servers in the overall MIMOSA architecture (see Figure 2), if required for the particular use case. The remote connectivity interface generally enables the remote application to access sensor data and aggregation functionality at the host terminal in the same manner local application would (with the difference of having the data being transmitted via cellular or other wireless networks).

The exact format and specification of the remote connectivity interface is defined in Section 8.

#### **4.8. Middleware Functionality**

The middleware implements commonly used functionality with respect to sensor provisioning, discovery and aggregation. It therefore enables reuse of this functionality by applications, simplifying the development of new applications. For that, it implements the functionality and the APIs as outlined in the appropriate work packages 2.2, 2.3, 2.4 and 2.5 and specified in work package 2.7.

The exact functionality of the middleware for the remote connectivity part of the architecture is presented in Section 8.

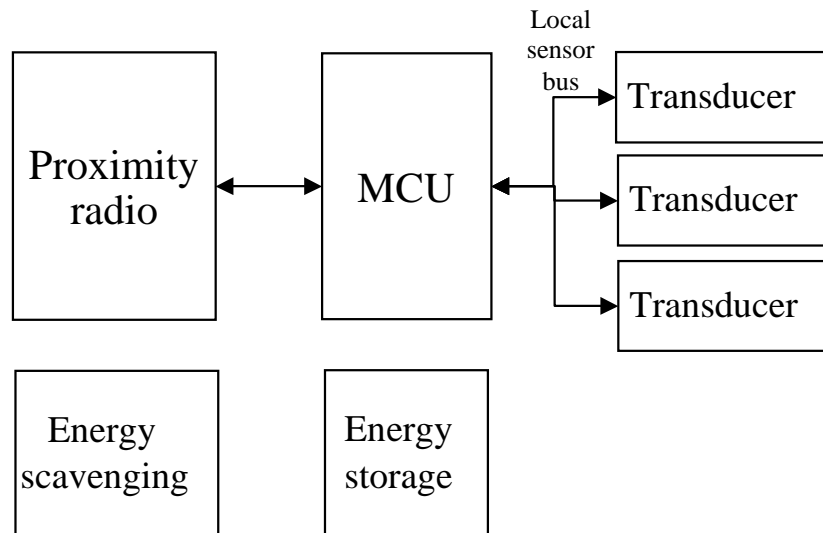
### **5. SENSOR RADIO NODE ARCHITECTURE**

Similarly to the terminal architecture, the high-level sensor radio node architecture is divided in to host architecture and interface architectures. The following interfaces are defined to be important for MIMOSA architecture: host - local connectivity, host – sensor, power interface, and energy scavenging interface.

#### **5.1. HOST ARCHITECTURE**

Host of the sensor radio node needs to be re-configurable so that all parties can build applications on top of it. It must also consume very little power. Therefore, core of the host is a low-power micro-controller unit. The focus of the work in MIMOSA project is not to develop low power micro-controllers, but to develop enabling platform for ambient intelligence. Therefore the host is build from existing components. The host should provide processing, memory and interface capabilities similar to the existing and near future platforms.

The host platform will be used for the sensor data processing and storage for the MIMOSA applications. The host platform will feature drivers to sensors, actuators and local connectivity module. Figure below shows the high level presentation of the sensor radio node.



**Figure 11: High-level presentation of the sensor radio node**

### 5.1.1. Hardware functionality

Radio sensor node is based to same MIMOSA architecture as terminal and there is processing unit, host, which is in this case low-power micro-controller. To enable flexible and powerful platform microcontroller should have at least properties that are mentioned in this specification.

Microcontroller unit (MCU) should have processor core that has enough computing power for applications, networking, controlling and measurement purposes. Power consumption of the micro-controller is essential to be as low as possible: microamperes in sleep mode and few milliamperes in full operation mode. There has to be enough memory, around 100 KB nonvolatile Flash-memory on board and more at external onboard memory. Amount of RAM memory has to be enough, but in this case it is not so important than systems with operating system.

Host has to have following interfaces: UART, SPI, I<sup>2</sup>C and several analog inputs with analog to digital converters that can be also external and JTAG for configuration and debugging purposes. RS-232 capable UART interface should also be available from connector from the board. Amount of free general purpose IO-pins should be at least ten, but these can be used to implement MIMOSA specific interfaces, like local connectivity interface (enhanced SPI) and power interface.

Hardware has to support all needed voltage levels from available power source. This main power source can vary but there has to be capability to implement power interface as specified to enable proper power supply supervision.

### 5.1.2. Software functionality

No operating system will be used.

The SW program is divided into initialization main loop and a clean up process. The main loop will periodically call sensor reading, sensor processing and connectivity functions.

Initial proposal for the main loop and how to share responsibilities among WP2 tasks are given in Figure 12.

```
While(1)
{
  networking_todo()
  /* Updates retransmission timers, if network layer ARQ is used,
  reads and handles incoming data. The handling of incoming data
  includes parsing the data for upper layer protocol, i.e, sensor
  protocol. Task 2.2 is responsible for the function content. */
  sensor_protocol_todo()
  /*Interprets incoming requests and generates responses. Task 2.3 is
  responsible for the function content, task 2.2 provides function calls
  for transmitting the responses*/
  sensor_reading_and_analysis_todo()
  /* Reads sensor values, produces meta-data from the read values.
  Task 2.4 is responsible for the function content. Task 2.3 provides
  function calls for reading and calibrating the raw sensor data.*/
  application_todo()
  /* Application specific routines. E.g. tuning the parameters related to
  local communication and sensor reading. The content will be fixed
  with WP1 */
}
```

Figure 12: Main loop of the sensor node host

### 5.2. Interface host – sensor

There is currently no standard interface to access different sensors. Convention how interfaces are defined varies from vendor to vendor. Many times vendors use standard interface like SPI that is then modified to their needs. Therefore several interfaces with modification possibility are defined initially that can be used to access sensors. Since this is a platform these interfaces are provided with external connectors. Figure below shows high-level presentation of the interfaces.

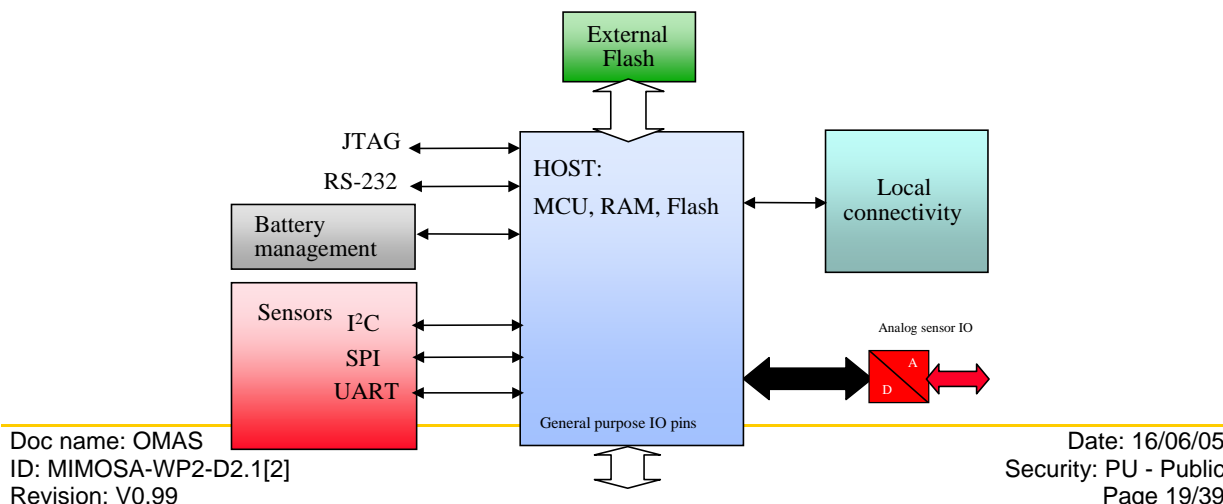


Figure 13: High-level presentation of interfaces

### 5.2.1. UART

#### Physical interface

#### Driver

Table 6: UART interface functions

Data Interface	Definition
S_uart_init(mode, baud_rate)	Initializes UART (mode= UART, SPI, I2C)
S_uart_disable(void)	Disables UART
S_uart_putchar(c)	Puts a character to UART TX buffer and sends it out
S_uart_putstr(s)	Sends a string out through UART, returns 1 if success
S_uart_getchar(void)	Reads an incoming character from UART buffer
S_spi_getchar(void)	Reads an incoming character in spi mode
__interrupt void uart0_tx_isr(void)	UART TX data buffer empty interrupt
__interrupt void uart0_rx_isr(void)	UART receive interrupt routine, executed when incoming symbols are in RX buffer

### 5.2.2. I2C

#### Physical interface

#### Driver

Table 7: I2C driver functions

Data Interface	Definition
S_init_I2C(void)	Routines for initializing the HW I2C interface
S_clock(void)	Send I2C clock to the bus
S_delay(void)	Delay function
S_wait (void)	Wait function
S_write_I2C(location, data)	This routine writes data to location using SW I2C implementation in microcontroller.
S_read_I2C(addr)	Reads data from address using I2C bus.

### 5.2.3. SPI

#### Physical interface

**Driver**

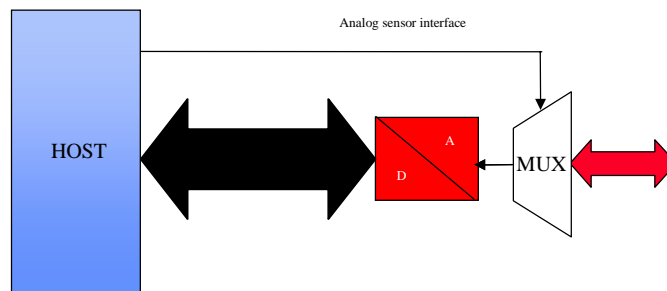
**Table 8: SPI driver functions**

S_uart1_init(mode, baud_rate)	Initializes SPI (mode= UART, SPI, I2C)
S_uart1_disable(void)	Disables SPI
S_uart1_putchar(c)	Puts a character to SPI TX buffer and sends it out
S_uart1_putstr(s)	Sends a string out through SPI, returns 1 if success
S_uart1_getchar(void)	Reads an incoming character from SPI
S_spi1_putchar(c)	Sends a character out to SPI bus
S_spi1_getchar(void)	Reads a character in from SPI bus
S_spi1_getint(channel)	Reads an integer in from SPI bus
__interrupt void uart1_tx_isr(void)	SPI TX data buffer empty interrupt
__interrupt void uart1_rx_isr(void)	SPI receive interrupt routine, executed when incoming symbols are in rx buffer

**5.2.4. Analog interface**

**Physical interface**

Several analog interfaces are available for sensors. Host has an AD conversion module and the analog inputs are multiplexed to AD converter. This block can be embedded straight to MCU chip or it can be external. Physical implementation differs, if this block is external, but driver functions have to be implemented as specified.



**Figure 14: Analog sensor interfaces**

**Driver**

**Table 9: AD converter block driver functions**

Data Interface	Definition
S_ad_init_sensor_n(void)	Configures AD channels for specified sensor_n measurement
S_ad_init_8(void)	Initialized AD converter for 8 channel

	measurement
S_adon (void)	Turn AD converter on
S_adoff (void)	Turn AD converter off
S_ad_start(void)	Start AD conversion
__interrupt void adc12(void)	AD converter interrupt (no interrupt/ADC overflow/ADC timing overflow)

### 5.3. Interface host –connectivity

#### Physical layer

Figure 15 presents physical layer interface architecture. The physical layer interface is identical to the interface inside the MIMOSA Terminal from the FPGA perspective.

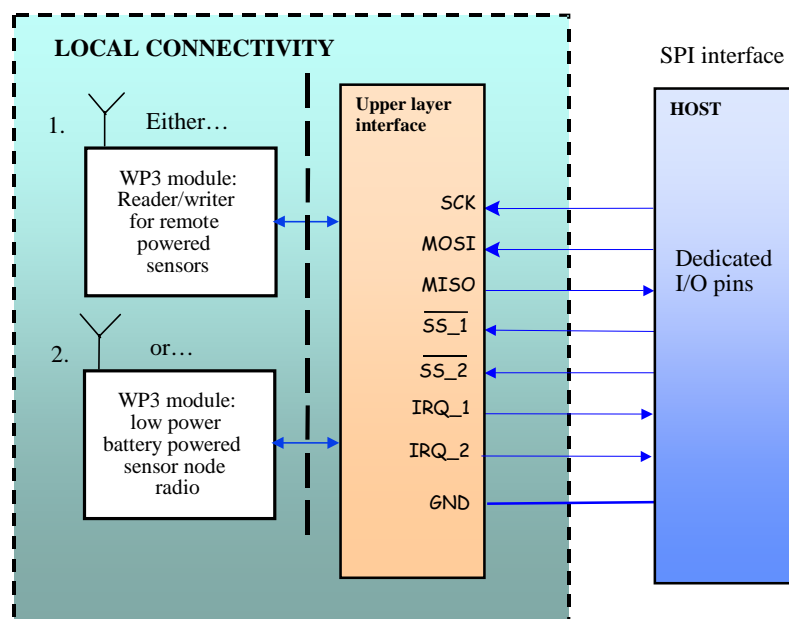


Figure 15: High-level MIMOSA local connectivity architecture for Radio sensor nodes

#### Drivers

The driver of MIMOSA local connectivity module controls the connectivity module. The driver provides the data exchange functionality and command interface for the sensor protocol and the applications.

Data Interface	Definition
C_Interface_read(TBA);	Must be called periodically from the main loop. The function detects whether the connectivity module has raised the interrupt. If so the data is read from the connectivity module and delivered to upper layer if not handled directly.
C_interface_write (TBA);	Write a specified amount of data from a specified memory location over the SPI

	interface.
--	------------

Command interface functions	Definition
C_advertise(TBA);	Makes local device visible to remote devices. I.e periodical RF activity.
C_advertise_stop(TBA);	Makes local device non-visible to remote devices.
C_scan(TBA);	Returns a list of visible remote devices.
C_scan_stop(TBA);	Interrupts remote device search
C_connect(TBA);	Creates a point-to-point connection with a remote device.
C_disconnect(TBA);	Disconnects the active connection.
C_accept(TBA);	Accepts a connection request from a remote device.

#### 5.4. Interface host – energy scavenging

Energy scavenging architecture will be derived later.

#### 5.5. Interface Battery - energy scavenging

Energy scavenging architecture will be derived later.

#### 5.6. Power interface

Radio sensor is battery-powered device. Main voltage supply, “battery”, can be either a fuel cell or ordinary Li-ion battery. In addition there is a possibility to switch to a secondary power source that can be used to charge battery, or super capacitor in case of fuel cells, instead of ordinary DC battery charger. This secondary power supply can be some energy-scavenging device, like solar cell or piezoelectric element that makes voltage from vibrations.

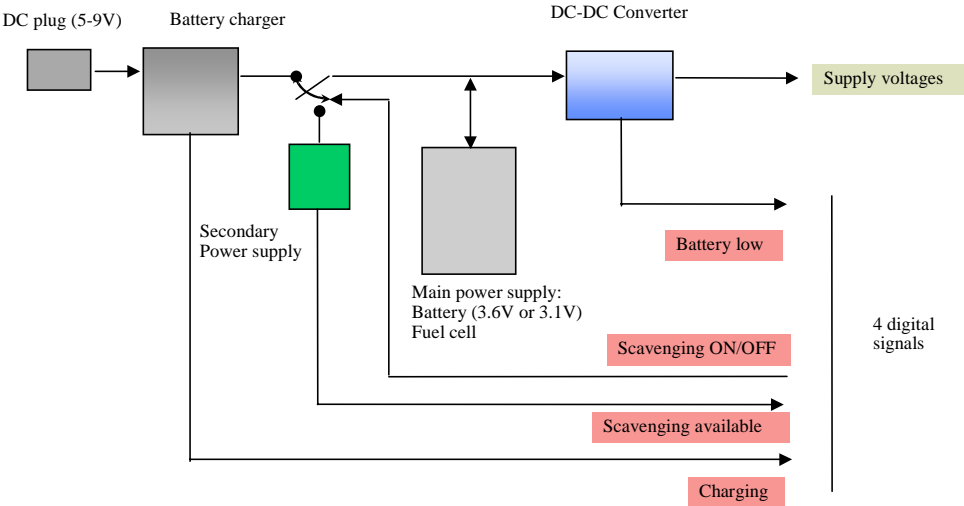
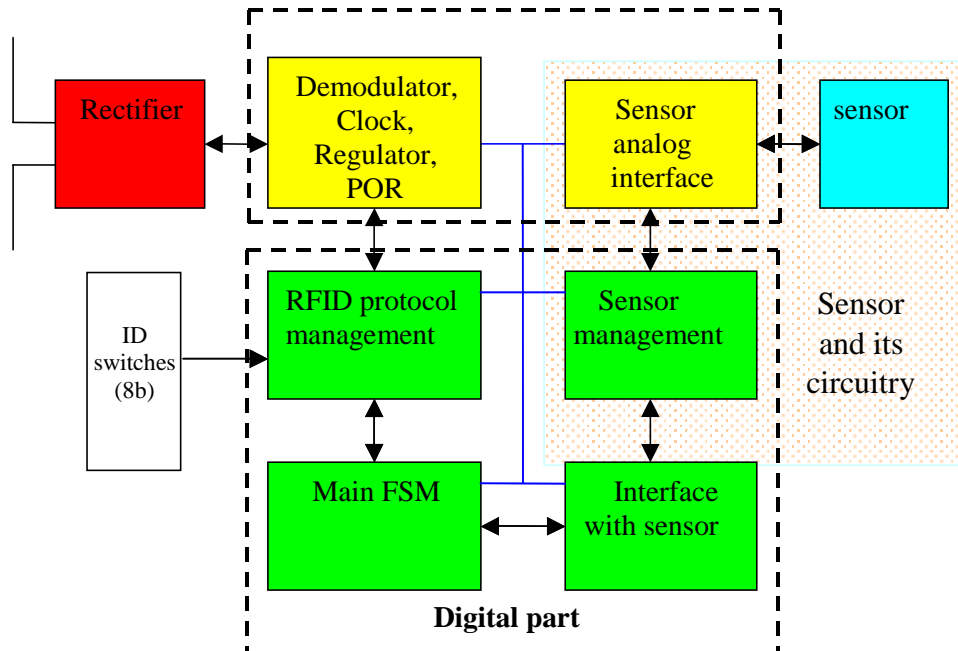


Figure 16:Radio sensor power interface

## 6. TAG ARCHITECTURE

### 6.1. RFID sensor architecture

The TAG architecture is presented in Figure 17.



**Figure 17: Tag architecture**

Antenna and rectifier forms the TAG front end. It includes also voltage limitation and backscattering. Analog part includes RFID analog circuitry (demodulator, clock, regulation, I/V references, POR) and analog sensor interface (sigma delta based).

Digital part includes RFID protocol management, state machine, sensor interface and sensor management (mainly filtering).

A standalone capacitive sensor will be used one.

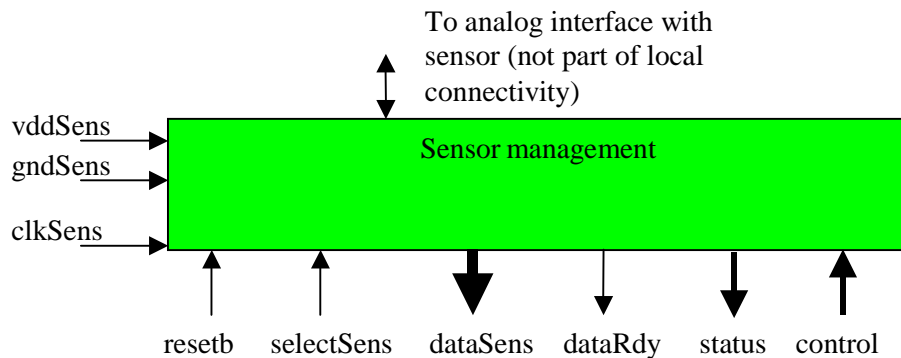
8 micro-switches are used to configure the TAG identifier.

Digital part will be implemented by means of FPGA (phase1A).

Concerning the analog part, details are available in the document **M3A2: Design specifications for WRPS**.

## 6.2. Sensor interface

Sensor interface specify the connection between sensor circuitry and RFID circuitry.



**Figure 18: Sensor interface**

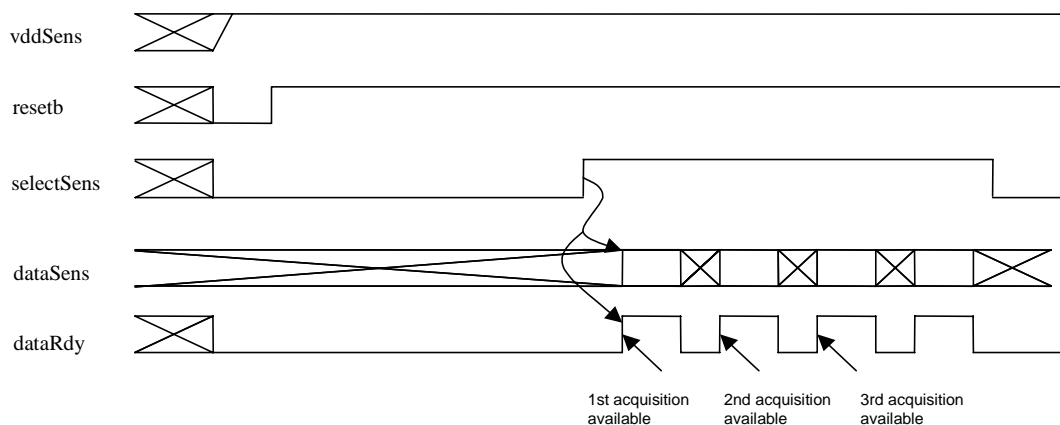
A parallel bus mechanism is used to insure communication between RFID circuitry and sensor circuitry. Wire name and directions (i for sensor input, o for sensor output) are:

- dataSens (o): measurement result (16b max)
- selectSens (i): if set to one sensor is on, otherwise sensor is switched off
- resetb (i): sensor circuitry reset (active low)
- dataRdy (o): set to one if measurement result is valid
- clkSens (i): sensor clock
- vddSens (i)
- gndSens (i)
- status (o) : sensor status (8b)
- control (i) : sensor control (8b)

A total of 38 wires are required (including sensor power supply).

## 6.3. Functional description

After a power on phase the RFID part is selected by means of a select(ID) sequence (see RFID sensor command set). Sensor is activated with a write command in the Dynamic Write memory allocation of the sensor.



---

Figure 19 Timing of sensor interface

#### 6.4. Sensor basic features

Sensor provided in phase 1a is a pressure sensor. It is included in a sigma-delta loop.

Reading time: 500ms

Sensor clock: 50kHz (TBC)

Regulated VddSens

Resolution: 10bits

Consumption: target is 20ua (sensor + associated circuitry)

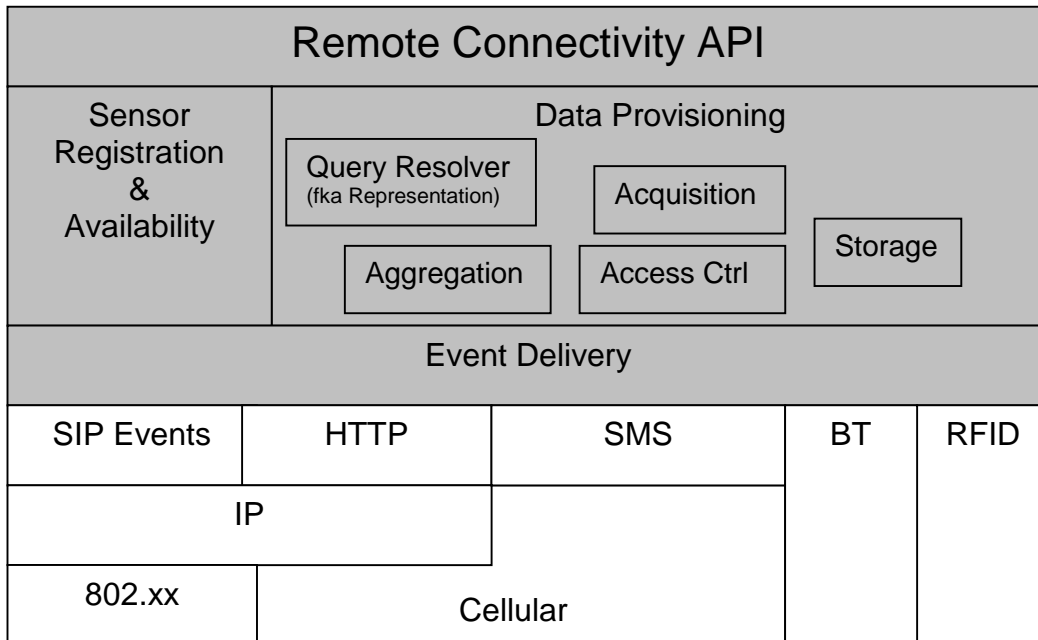
#### 7. REFERENCES

- [1] OMAP1510 Multimedia Processor Technical Reference Manual Literature Number: SWPU030A June 2002. [www.ti.com](http://www.ti.com)
- [2] Nieminen Heikki (editor) "Intelligent sensor architecture phase I" MIMOSA deliverable D2.3
- [3] Lappeteläinen Antti (editor) "Local connectivity; Architecture Specification phase I" MIMOSA deliverable D2.2
- [4] A. Roach, "SIP-Specific Event Notification", RFC 3265, July 2002
- [5] J. Rosenberg, "The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)", Internet Draft, Internet Engineering Task Force, (work in progress), May 2003

## 8. Appendix: Remote Connectivity Middleware Specification

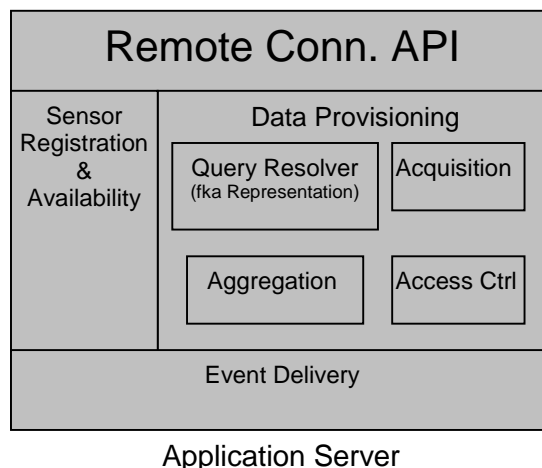
### 8.1. Functionality of Each Component

The functionality of each middleware component (see Figure 20 below) is specified in the following subsections.



**Figure 20: Middleware Components**

This functionality is realized within a distributed architecture, as outlined in Figure 2. While the MIMOSA terminal needs to implement all components of Figure 20, the application server implements only part of the functionality, as shown in Figure 21.



**Figure 21: Middleware Components at Application Server**

The following sections will give some more details on which functionality needs to be provided by each of the components, in addition to implementing the appropriate interfaces as defined in Section 8.2.

### 8.1.1. Event Communication Paradigm in REMONT

Before we introduce the functionality of each component of the middleware, some words on the underlying communication paradigm. Many interfaces within the middleware are based on a common event delivery paradigm with the following characteristics:

- The event delivery is based on individual *subscriptions* to so-called *events*. These events usually represent state information associated to certain resources.
- In order to allow for proper identification of the particular subscription, allowing for several subscriptions at any given time, there is a *dialog identifier* provided with each confirmation of the subscription.
- The semantic of each event is application-specific and based on a certain semantic description.
- The event delivery allows for arbitrary payload for each event subscription and notification
- For each subscription, there exists a subscriber (client) and an event server
- There exists five methods (messages)
  - SUBSCRIBE allows for initiating a subscription. It contains
    - *event name*,
    - *event-specific content* (such as query or filter descriptions)
    - *lifetime* of the subscription. If the lifetime equals zero, the subscription serves as a “one-shot” subscription or poll of information, i.e., there will be only a single notification and there is no subscription dialog established. Otherwise, the lifetime parameter specifies the duration of the subscription in seconds.
    - *FROM* and *TO* parameters, identifying the subscriber and event server. These identifiers depend on the particular bearer of information (such as SMS, IP, SIP events, HTTP URIs) on which the event communication paradigm will be implemented.
  - PUBLISH allows for publishing relevant information for a particular event, such as state information or information used in the state determination by the event server. The sender of the PUBLISH method is usually not the subscriber to the event (but could be), the receiver is the event server that hosts the event. The method contains
    - the *event name* to which the information relates
    - *event-specific content*, i.e., the information relating to the event.
    - *FROM* and *TO* parameters, identifying the publisher and event server. These identifiers depend on the particular bearer of information (such as SMS, IP, SIP events, HTTP URIs) on which the event communication paradigm will be implemented.
  - CONFIRM always confirms each publication to the publisher and each subscription to the subscriber. The confirmation can either be positive (subscription granted) or negative. In the latter case, a reason code is provided. In the positive case for a subscription, the *dialog identifier* is provided to the subscriber to identify the subscription dialog. For a publication, there is no dialog established.
  - NOTIFY allows for sending notifications for a particular subscription. These notifications are triggered depending on the particular event

semantic. Examples are state changes in the event or other reasons. There is always an initial NOTIFY, conveying the initial state of the subscription right after the subscription has been granted (note that this initial NOTIFY is not sent if the submission has not been granted, indicated through a negative CONFIRM). If the lifetime of the subscription is non-zero, future NOTIFY messages are possible, depending on the semantic of the event subscription. The NOTIFY method includes *FROM*, *TO*, and *event name* fields from the original subscription. It further includes the *dialog identifier* to properly relate notification and subscription.

- BYE explicitly terminates a subscription. The dialog identifier needs to be given in order to relate the BYE method to the particular subscription. A BYE method can be sent from either the subscriber or the event server. In the latter case, a reason code is provided.

The above outlined event delivery paradigm is very similar to the SIP event framework [4]. Interfaces are defined throughout the middleware some of which are based on the outlined event communication paradigm. These interfaces implement the functionality as described in this section. Only additional, interface-specific functionality will be described for these interfaces. We will refer appropriately to the current section in such cases.

### 8.1.2. Event Delivery Component

This component implements the event communication paradigm of Section 8.1.1 in order to provide a general purpose component for all remote communication between application server and MIMOSA terminal.

At the application server and MIMOSA terminal, there exist subscriber/publisher as well as event server functionality, while the code repository merely acts as a publisher of information.

The functionality is provided through interface  $E_D$  (see Section 8.2.4 for remote communication between event delivery components) and interface  $E_{D(local)}$  (see Section 8.2.9 for local communication from other components towards the local event delivery component). With respect to the actual functionality of this component, see also Section 8.1.1.

Incoming requests, such as SUBSCRIBE at the event server side or NOTIFY at the subscriber side, are dispatched to the appropriate local component, based on the event information given in the request (see interface descriptions in Section 8.2 for the event information for each middleware component). In this dispatching, the particular middleware component is also provided with the information included in the request, i.e., event name, event body, dialog identifier, and FROM/TO information.

The event delivery component implements bearer-specific functionality to provide the interface  $E_D$  to upper layer components. Currently supported is TCP/IP as a transport bearer.

### 8.1.3. Acquisition Component

On the application server side, the acquisition component is one of the two components to be accessed by the application(s). This access might happen locally (i.e., application resides locally on the application server) or remotely (e.g., through providing CORBA-enabled access to the component) and is defined through Interface  $RAP_{AA}$  (see Section 8.2.1).

Acquisition requests are pre-processed at the application server by

- checking the access rights of the particular application with respect to the requested data (interface  $A_{ctrl(local)}$  as defined in Section 8.2.11) and
- verifying the availability of particular sensors that are required to fulfill the request (interface  $A_{RA(local)}$  as defined in Section 8.2.10)
- verifying the availability of code for particular aggregation functionality, if required in the request, (interface  $A_{RA(local)}$  as defined in Section 8.2.10)

Hence, in this case the application acts as a subscriber while the application server implements the event server, the event being the acquisition request.

If these verifications return positively, the acquisition request is made remotely at the appropriate MIMOSA terminal through the interface  $A_C$  as defined in Section 8.2.6. In this case the application server acts as a subscriber for an event (being the acquisition request) at the MIMOSA terminal, the terminal's acquisition component acting as a event server.

Acquisition requests at the MIMOSA terminal are received remotely (through interface  $A_C$ ). The query of the request is forwarded to the query resolver component (see Section 8.1.4) through interface  $A_{Q(local)}$  as defined in Section 8.2.13. The query resolver returns pointers to sensor objects and (if required) aggregation objects required to satisfy the query.

Based on the return results of the query resolver component, the required sensor data is acquired via interface  $SSI_A$  (see Section 8.2.7), using the returned sensor objects.

If required, obtained sensor data is aggregated according to the query (implemented through communicating with the aggregation component via interface  $A_{A(local)}$  as defined in Section 8.2.12). If the request is fulfilled, appropriate notifications are fired via interface  $A_C$ .

Note that several acquisition requests may be pending at either the application server but also at the MIMOSA terminal. Hence, there is functionality required to support such multiple requests.

#### **8.1.4. Query Resolver Component**

Incoming acquisition requests contain queries for sensor data (or aggregations of sensor data). These queries are based on a defined abstraction model for the sensor data (and its aggregations). Such abstraction model will be defined as part of work package 2.4 in MIMOSA.

The query resolver component implements functionality to

- parse the incoming request, based on the abstraction model and the query language syntax
- determine the sensor data to be acquired by the acquisition component
- determine the aggregation functionality that is required in case when aggregated data is requested.

While the first bullet items merely aims at the validity of the request, the last two ones aim at instructing the acquisition component as to what sensor data to acquire and what aggregation component functionality to use in order to fulfill the request.

For that, the query is received and the results are returned via the interface  $A_{Q(local)}$  (as defined in Section 8.2.13).

### 8.1.5. Aggregation Component

The aggregation component implements functionality for the acquisition component to support aggregated data.

Based on the acquisition query, the query resolver determines the required aggregation functionality (see Section 8.1.4). The acquisition component then requests the determined aggregation functionality from the aggregation component (via interface  $A_{A(local)}$  as defined in Section 8.2.12). If the required functionality does not exist, the aggregation component returns a negative result, i.e., the acquisition is rejected.

During the acquisition process, the acquisition component provides the aggregation component with the appropriate sensor data to perform the desired aggregation functionality (via interface  $A_{A(local)}$  as defined in Section 8.2.12). If obtained sensor data is required for future use in order to perform the aggregation, it can be stored with the local storage component (via interface  $A_{S(local)}$  as defined in Section 8.2.14).

### 8.1.6. Access Control Component

This component only resides on the application server side. As explained in Section 8.1.3, the access control component is consulted in case of an incoming acquisition request in order to determine appropriate access rights for the particular application with respect to the particular data requested (using interface  $A_{ctrl(local)}$  as defined in Section 8.2.11).

Although depicted as a local component in Figure 21, it is possible to implement the access control component in an outside server. This server might host other access control policies as well. For instance, the additional server might be compliant to XCAP [5], i.e., the interface  $A_{ctrl(local)}$  as defined in Section 8.2.11 would be compliant to XCAP, the policies would be defined as *XCAP usages*.

### 8.1.7. Storage Component

The storage component implements the temporary storage of acquired data, e.g., for implementing certain aggregation functionality.

This storage functionality uses simple memory heap functionality in the MIMOSA terminal, assuming no existing hard-drive functionality. However, one could also envision the usage of externally available memory cards (e.g., MMC).

### 8.1.8. Registration & Availability Component

The application server's R&A component serves as a central registry for this purpose, implementing an event server, used by the local acquisition module. The following information is published to the R&A component at the application server:

- Available sensors at the intermediary gateway. For this, the R&A component at the MIMOSA terminal subscribes to the sensor capabilities using interface  $SSI_R$  as defined in Section 8.2.8. Obtained sensor information is then published at the application server's R&A component using interface  $R_A$  as defined in Section 8.2.5.
- Available aggregation functionality at the MIMOSA terminal. For this, the R&A component at the MIMOSA terminal registers all available functionality with the application server's R&A component.

The available information at the R&A component at the application server can either be queried by the application directly, using interface  $RAP_{ARA}$  as defined in Section 8.2.1, or is queried by the acquisition component, using interface  $A_{RA(local)}$  as defined in Section 8.2.10, before sending an acquisition request to the MIMOSA terminal.

It is important to note that the R&A component also provides functionality to subscribe to the availability of sensors (or aggregation functionality).

## 8.2. Interfaces Between the Components

The following chapter describes the interfaces and protocols of the remote connectivity part of the middleware.

### 8.2.1. The RAPI Interface (Remote Connectivity API)

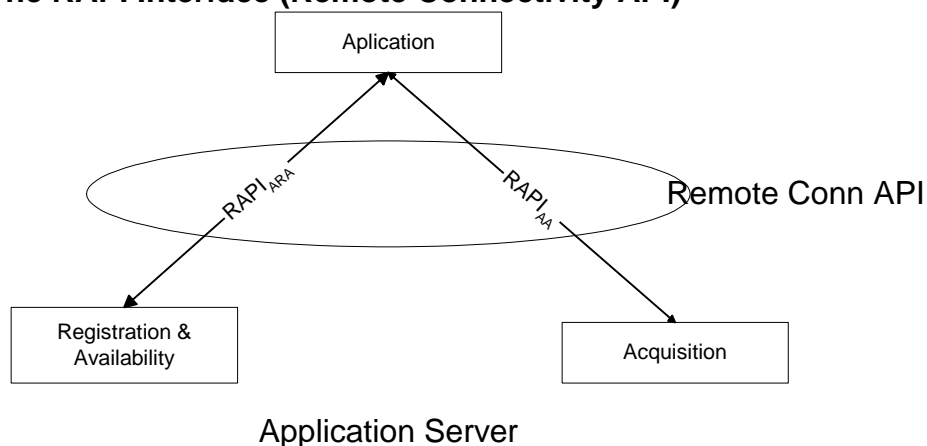


Figure 22: The RAPI Interface

The RAPI interface constitutes the Remote Connectivity API in Figure 20. It is divided into two parts, namely the RAPI<sub>ARA</sub> and the RAPI<sub>AA</sub> interfaces.

### 8.2.2. The RAPI<sub>ARA</sub> interface

The RAPI<sub>ARA</sub> interface allows remote applications for subscribing to the availability of sensors or aggregation functionality within the system. This interface is based on the event delivery paradigm outlined in Section 8.1.1:

- The application is the subscriber, the R&A component is the event server
- The event name is '*available*', the event payload of the SUBSCRIBE method carries the description for the sensor or aggregated sensor
- If the lifetime is zero, the subscription constitutes a normal discovery of available sensors, while a lifetime of non-zero constitutes a subscription to current and future availability of sensors.
- The FROM field contains the application identifier, while the TO field contains the application server identifier.
- The NOTIFY methods returns the matching sensors, based on the provided semantic of the subscription semantic. The initial NOTIFY contains the currently available sensors, while future NOTIFY methods will contain then available sensors. Every plain sensor description additionally includes an identifier of the MIMOSA terminal at which the sensor is registered.

Techniques such as defined in [4] are candidates for implementing this part due to the similarities to plain SIP events.

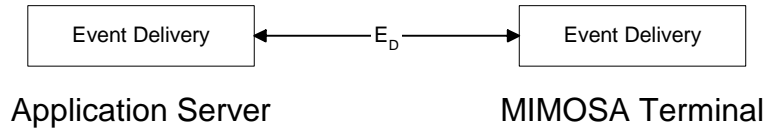
### 8.2.3. The RAPI<sub>AA</sub> interface

The RAPI<sub>AA</sub> interface allows for instructing the acquisition component to acquire (aggregated or plain) sensor data from a particular MIMOSA terminal. Note that the pre-requisite functionality, as described in Section 8.1.3, is executed before the acquisition subscription is made. This interface is based on the event delivery paradigm outlined in Section 8.1.1:

- The application is the subscriber, the acquisition component is the event server
- The event name is '*acquire*', the event payload of the SUBSCRIBE method carries the query for the (aggregated or plain) sensor data
- If the lifetime is zero, the subscription constitutes a simple query for particular (aggregated or plain) sensor values, while a lifetime of non-zero constitutes a subscription to current and future values.
- If the application wants to acquire sensor data from a particular MIMOSA terminal, the identifier of this MIMOSA terminal needs to be given in the event payload. This identifier depends on the particular bearer of information supported by the event delivery component.
- The FROM field contains the application identifier, while the TO field contains the application server identifier.
- If the acquisition request is invalid, e.g., due to lack of access rights, non-availability of aggregation functionality or other, the CONFIRM method will return an appropriate error reason code.

- The NOTIFY methods returns the current values based on the original query. The body of the NOTIFY method also includes an identifier of the MIMOSA terminal at which the values were acquired.

#### 8.2.4. The $E_D$ Interface



**Figure 23: The  $E_D$  Interface**

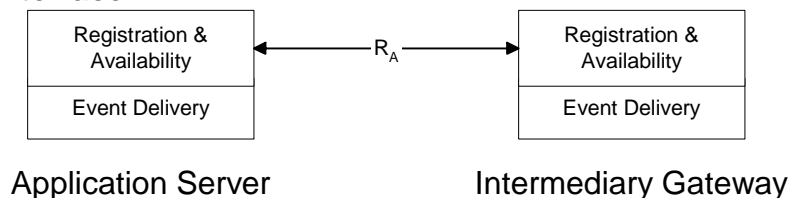
The  $E_D$  interface provides the event delivery functionality as described in Section 8.1.1 for the remote case, i.e., for communicating between application server and MIMOSA terminal. The parameters described in Section 8.1.1 are therefore provided by this interface.

All communication between remote components happens via the  $E_D$  interface, together with using the generic  $E_{D(local)}$  template interface. In other words, any particular interface  $X$  is implemented by conveying the parameters to the event delivery component through the  $E_{D(local)}$  template interface. The event delivery component in turn provides the delivery of the parameters to the appropriate component according to the definition of interface  $X$ , either locally through the  $E_{D(local)}$  interface or remotely through the  $E_D$  interface plus a local dispatching via the  $E_{D(local)}$  interface.

Since the  $E_D$  interface is the basic remote interface, its establishment has to take into account the particular problem of NAT/firewall traversal in mobile environments. In other words, the interface needs to be established from the MIMOSA terminal towards the application server in the presence of these middleboxes.

The event delivery component provides appropriate mappings of the interface onto different bearers of information. For SMS, HTTP, and pure IP bearers, this mapping needs to be defined while for SIP events, this mapping happens onto appropriate SIP event packages.

#### 8.2.5. The $R_A$ Interface



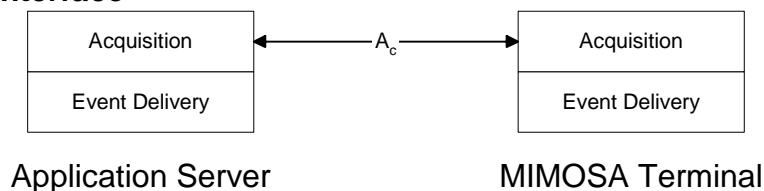
**Figure 24: The  $R_A$  Interface**

The  $R_A$  interface allows for publishing the availability of sensors with the system. This interface is based on the event delivery paradigm outlined in Section 8.1.1:

- The R&A component at the MIMOSA terminal is the publisher, the R&A component at the application server is the event server

- The event name is 'available', the event payload of the PUBLISH method carries the description for the sensor(s)
- It is possible to publish information about several sensors within a single operation by appropriately building descriptions for each sensor, carried as a single body in the publication
- The FROM field contains the identifier of the MIMOSA terminal, the TO field contains the application server identifier

### 8.2.6. The A<sub>c</sub> Interface

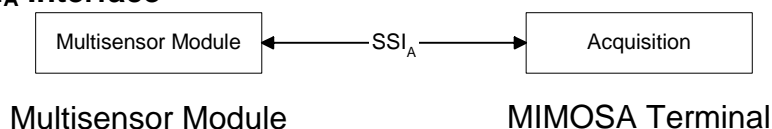


**Figure 25: The A<sub>c</sub> Interface**

The A<sub>c</sub> interface allows for acquiring (aggregated or plain) sensor data from a particular MIMOSA terminal. Note that the pre-requisite functionality, as described in Section 8.1.3, is executed before the acquisition subscription is made. This interface is based on the event delivery paradigm outlined in Section 8.1.1:

- The acquisition component at the application server is the subscriber, the acquisition component at the MIMOSA terminal is the event server
- The event name is 'acquire', the event payload of the SUBSCRIBE method carries the query for the (aggregated or plain) sensor data
- If the lifetime is zero, the subscription constitutes a simple query for particular (aggregated or plain) sensor values, while a lifetime of non-zero constitutes a subscription to current and future values.
- The FROM field contains the application server identifier, the TO field contains the identifier of the MIMOSA terminal
- If the acquisition request cannot be fulfilled, e.g., due to lack of resources, the CONFIRM method will return an appropriate error reason code.
- The NOTIFY methods return the current values based on the original subscription.

### 8.2.7. The SSI<sub>A</sub> Interface



**Figure 26: The SSI<sub>A</sub> Interface**

The SSI<sub>A</sub> interface allows for acquiring sensor data from a particular multisensor module. The API for this interface is the Sensor API as described in WP2.3, the protocol chosen is the SSI protocol [3].

### 8.2.8. The $SSI_R$ Interface

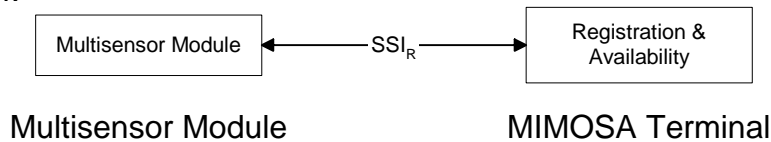


Figure 27: The  $SSI_R$  Interface

The  $SSI_R$  interface allows for subscribing and querying the capabilities of a multisensor module. The API for this interface is the Sensor API as described in WP2.3, the protocol chosen is the SSI protocol [3].

### 8.2.9. The $E_{D(local)}$ Interface

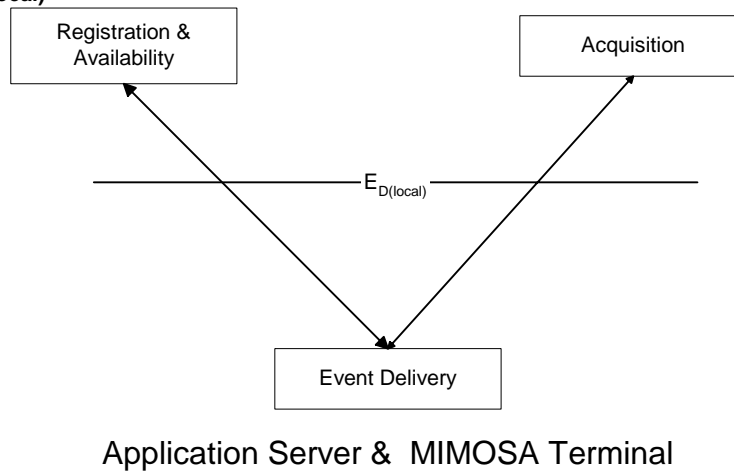


Figure 28: The  $E_{D(local)}$  Interface

The  $E_{D(local)}$  interface serves as a “template interface”, providing locally the event delivery functionality as described in Section 8.1.1. The parameters described in Section 8.1.1 are therefore provided by this interface.

For any communication between components that is based on the event delivery paradigm as described in Section 8.1.1, a particular interface  $X$  is implemented through conveying the parameters to the event delivery component via the  $E_{D(local)}$  template interface. The event delivery component in turn provides the delivery of the parameters to the appropriate component according to the definition of interface  $X$ , either locally through the  $E_{D(local)}$  interface or remotely through the  $E_D$  interface plus a local dispatching via the  $E_{D(local)}$  interface.

### 8.2.10. The $A_{RA(local)}$ Interface

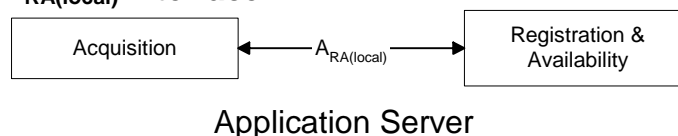


Figure 29: The  $A_{RA(local)}$  Interface

The  $A_{RA(local)}$  interface allows for querying the availability of certain sensor and aggregation functionality.

For that, the sensor and aggregation functionality description is provided to the R&A component, which returns Boolean values as results. Several sensors and aggregation functionalities can be checked, resulting in a Boolean result vector.

### 8.2.11. The $A_{ctrl(local)}$ Interface



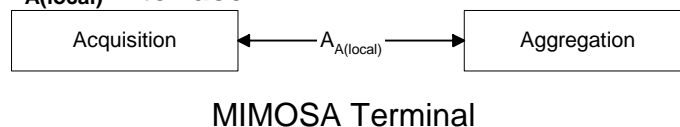
**Figure 30: The  $A_{ctrl(local)}$  Interface**

The  $A_{ctrl(local)}$  interface allows for querying whether or not a certain identity has proper access rights for particular sensor and aggregation functionality.

For that, the identity is given as a parameter together with a pre-defined identifier for the resource, i.e., the sensor or aggregation functionality. The query is returned with a Boolean result.

Although depicted as a local interface, it is possible that  $A_{ctrl(local)}$  is implemented as a remote interface. This is for cases in which the access control component resides on an outside server (see also Section 8.1.6). This server might host other access control policies as well. For instance, the additional server might be compliant to XCAP [5], i.e., the interface  $A_{ctrl(local)}$  would then be compliant to XCAP, the policies would be defined as *XCAP usages*.

### 8.2.12. The $A_A(local)$ Interface

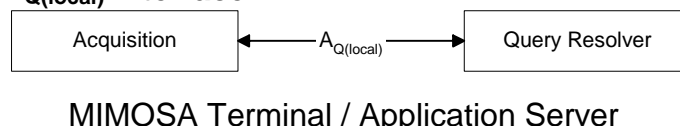


**Figure 31: The  $A_A(local)$  Interface**

The  $A_A(local)$  interface allows for initiating the aggregation of particular sensor data.

For that, the sensor objects and the particular aggregation object, obtained at the acquisition component during the query resolving, are used as input parameters. The aggregation is fed into the aggregation object and returned.

### 8.2.13. The $A_Q(local)$ Interface



**Figure 32: The  $A_Q(local)$  Interface**

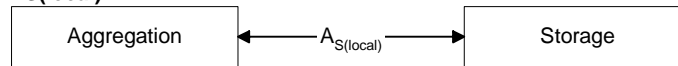
The  $A_Q(local)$  interface allows for initiating the resolving of the acquisition query at the MIMOSA terminal as well as at the application server (see also Section 8.1.3 and 8.1.4).

For that, the acquisition query is provided as an input parameter.

At the application server, the query resolver returns identifiers for the required sensors and aggregation functionalities to satisfy the query, or the resolver returns an error code. These identifiers will be used to verify access rights and availability as described in Section 8.1.3.

At the MIMOSA terminal, the query resolver returns pointers to objects for the required sensors and aggregation functionalities to satisfy the query, or the resolver returns an error code.

#### 8.2.14. The $A_{S(local)}$ Interface



MIMOSA Terminal

**Figure 33: The  $A_{S(local)}$  Interface**

The  $A_{Q(local)}$  interface allows for storing and retrieving sensor data for future use in order to perform the desired aggregation (see also Section 8.1.5).

For that, the sensor data is written to the storage component using the current value and a sensor identifier. The storage component returns an identifier for future retrieval.

Upon future retrieval, this identifier is used to retrieve the data. An additional Boolean value determines whether or not to delete the stored value from the storage. The sensor value and sensor identifier are returned to the aggregation component.